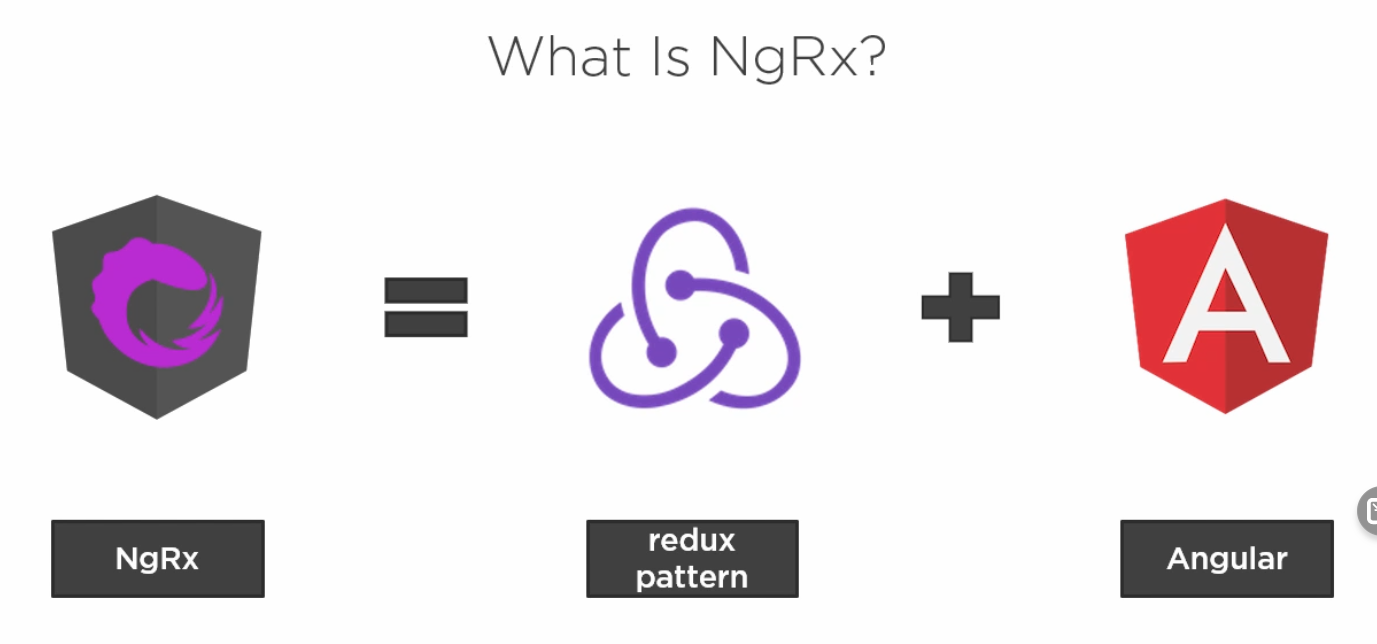
NgRx

# Redux

## Intro



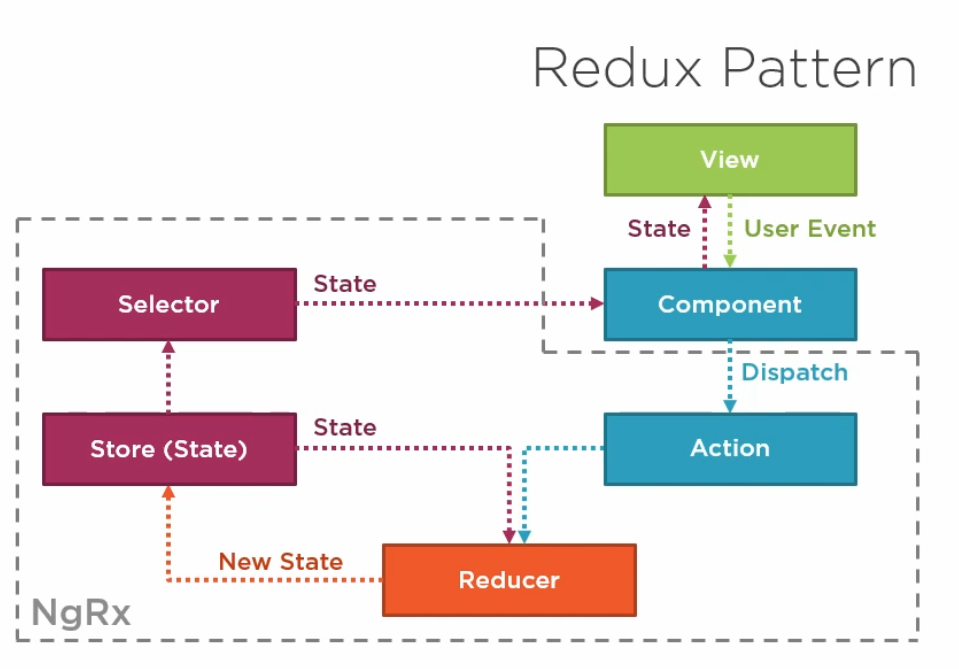
Redux flow is one way communication from component to store

The component dispatches an Action to a Reducer which can access the state and update it.

The state is immutable which means the Reducer always creates a new state and never modifies the existing state

Any component needs to subscribe to the store using Selector to get notified of any changes in the Store

The Selector knows how to locate and return data from the Store



## Why do we use Redux?

To reduce the overhead of components communication and passing data between components

If a user navigated away for a second and came back to the same page, the component will have to make http requests again to retrieve all the data for that page

NgRx Store provides Client-side cache



## Redux principles

Singe Source of Truth is the Store

State is read-only and changed by dispatching Actions

Changes are made using Pure functions called Reducers

## When to use Redux for state management?

* When you build an application with a lot of user interactions and multiple data sources.
* When managing state in services are no longer sufficient.

### Shari principle

* Shared: state that is accessed by many components and services.
* Hydrated: state that is persisted and rehydrated from external storage.
* Available: state that needs to be available when re-entering routes.
* Retrieved: state that must be retrieved with a side-effect.
* Impacted: state that is impacted by actions from other sources.

# Store

It is a JS Object which contains all of the state shared across the Application

## What shouldn’t be in the store

Unshared state which is only relative to that component

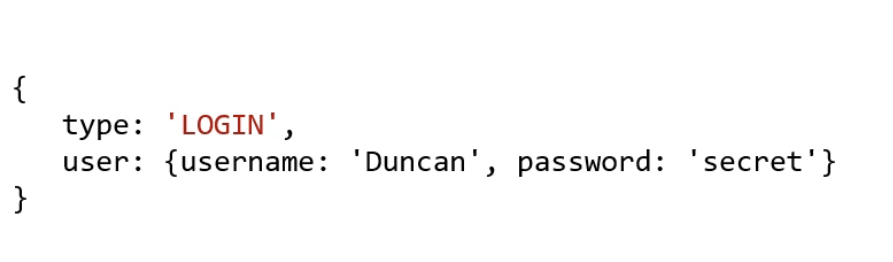
Angular Form state

Non serializable state which cant be serialized to json object

# Actions

All user events are dispatched as Actions which affect the Store

Actions express unique events that happen throughout your application.



Not all actions mutate the store via Reducers. Because some Actions have side effects which are managed by NgRx Effects library

## Writing Acitons:

* Upfront - write actions before developing features to understand and gain a shared knowledge of the feature being implemented.
* Divide - categorize actions based on the event source.
* Many - actions are inexpensive to write, so the more actions you write, the better you express flows in your application.
* Event-Driven - capture events not commands as you are separating the description of an event and the handling of that event.
* Descriptive - provide context that are targeted to a unique event with more detailed information you can use to aid in debugging with the developer tools.

## Example:

[Login Page] Login

* The category of the action is captured within the square brackets [].
* The category is used to group actions for a particular area, whether it be a component page, backend API, or browser API.
* The Login text after the category is a description about what event occurred from this action.

# Reducers

They are responsible for transitioning the Application from one State to Another. Reducer functions handle these transitions by determining which [actions](https://ngrx.io/guide/store/actions) to handle based on the action's type.

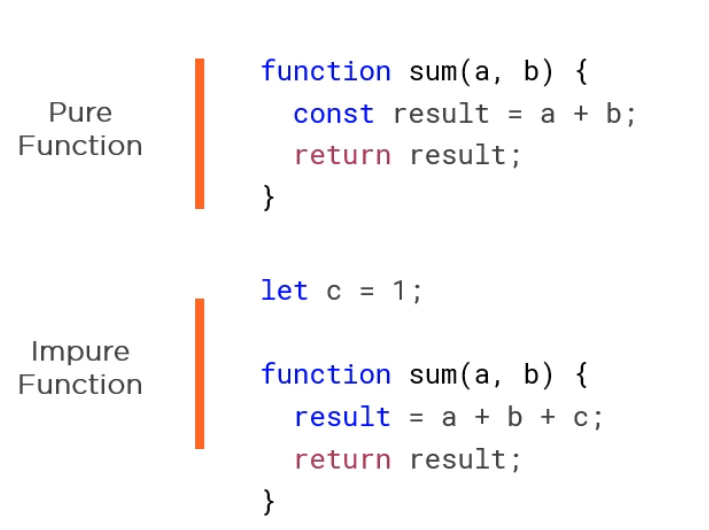
Each reducer function takes the latest [Action](https://ngrx.io/api/store/Action) dispatched, the current state, and determines whether to return a newly modified state or the original state.

## Pure Function

A function that doesn’t have any side effects on input or global objects.

A function that given the same input will have the same output

It should not depend on outside variables

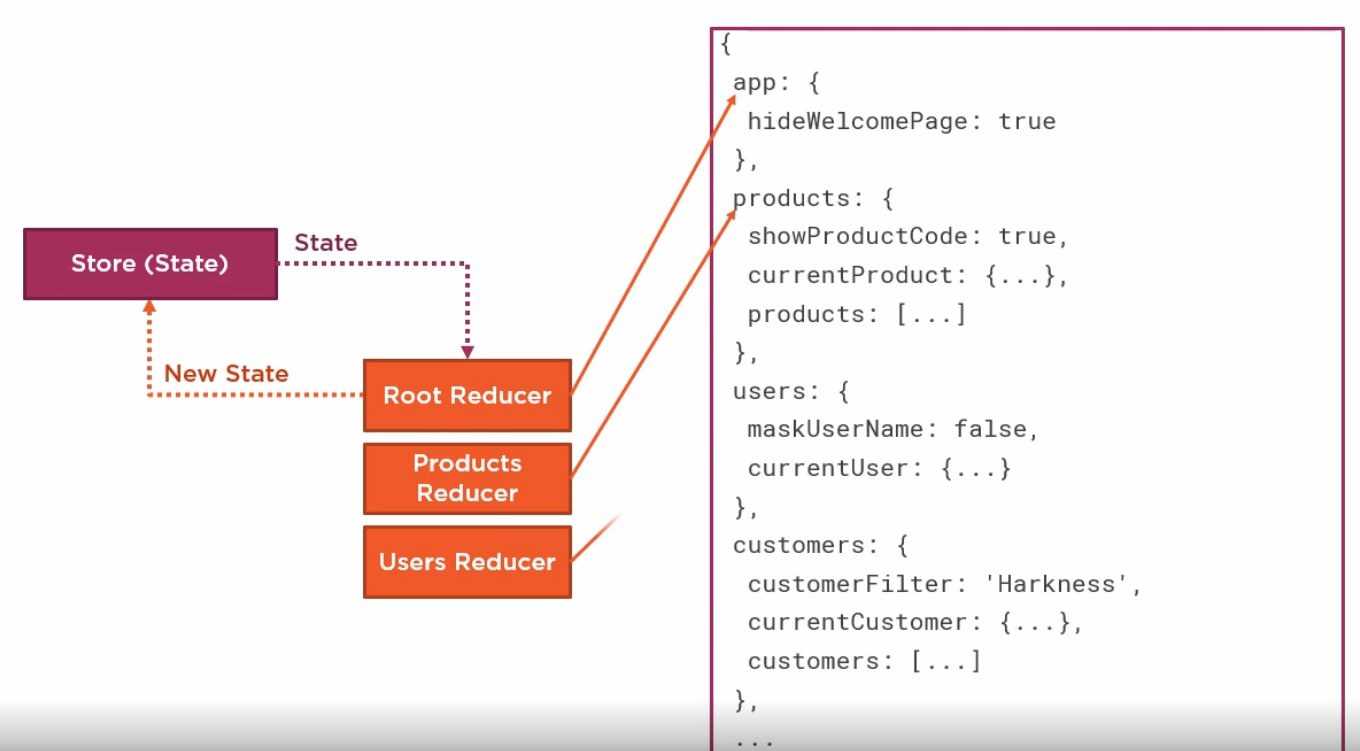


There are a few consistent parts of every piece of state managed by a reducer.

* An interface or type that defines the shape of the state.
* The arguments including the initial state or current state and the current action.
* The functions that handle state changes for their associated action(s).

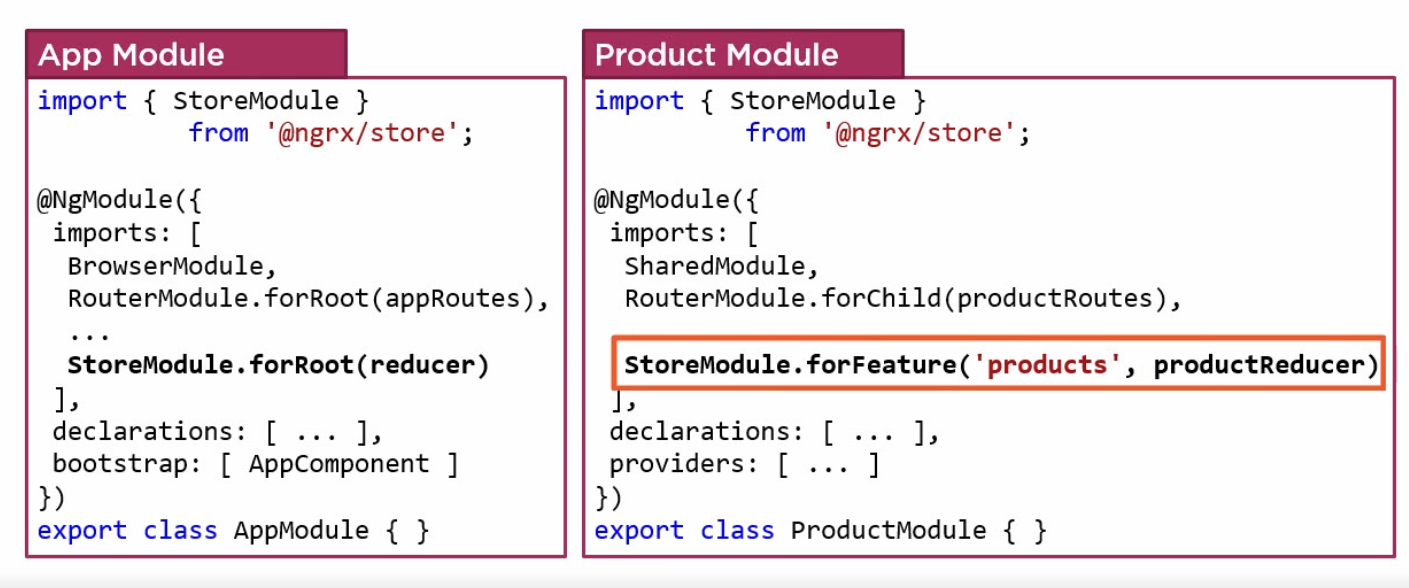
# Reducers per Feature

In order to divide or organize our store with a logical hierarchy similar to the feature modules hierarchy we can use multiple reducers per feature module

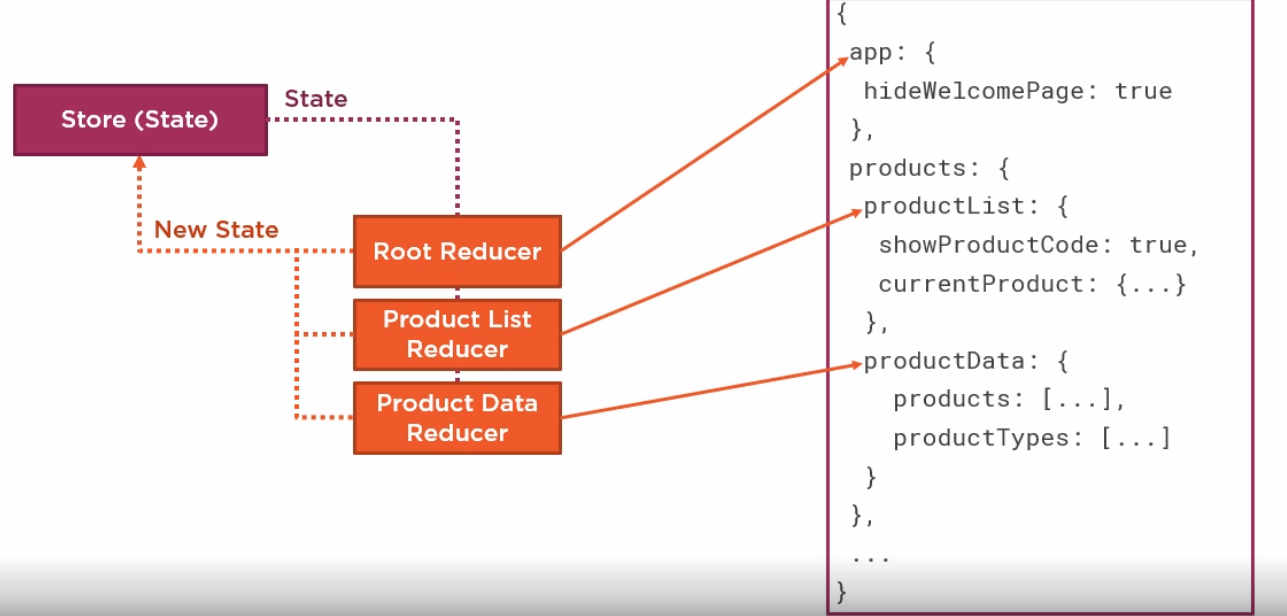


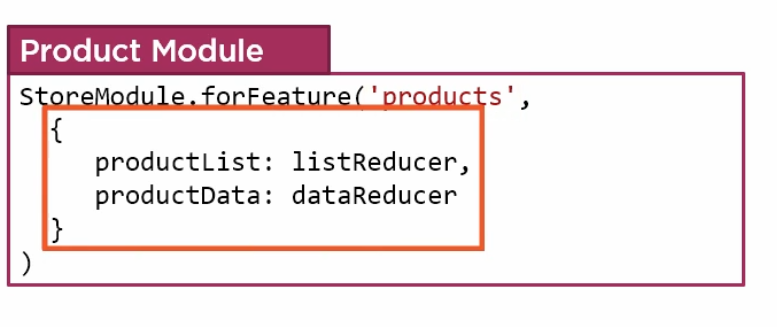
State isn’t created for a module that isn’t loaded

To register it; it is similar to RouterModule lazy loading registration



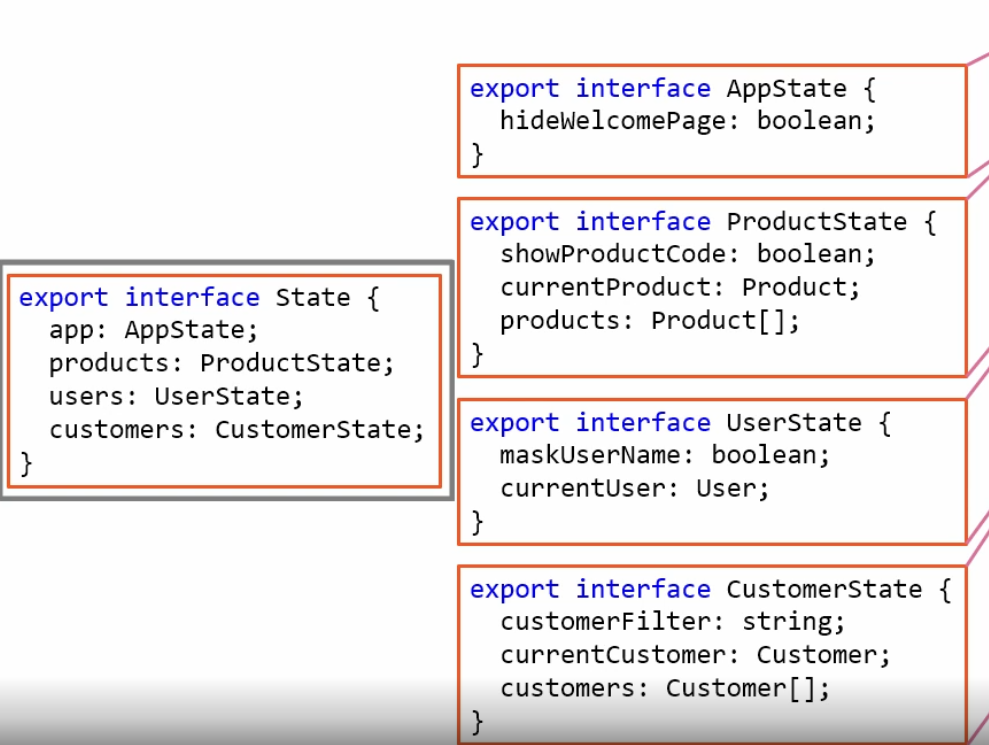
If we want to break the reducers even further inside each feature module, have multiple reducers for one feature module



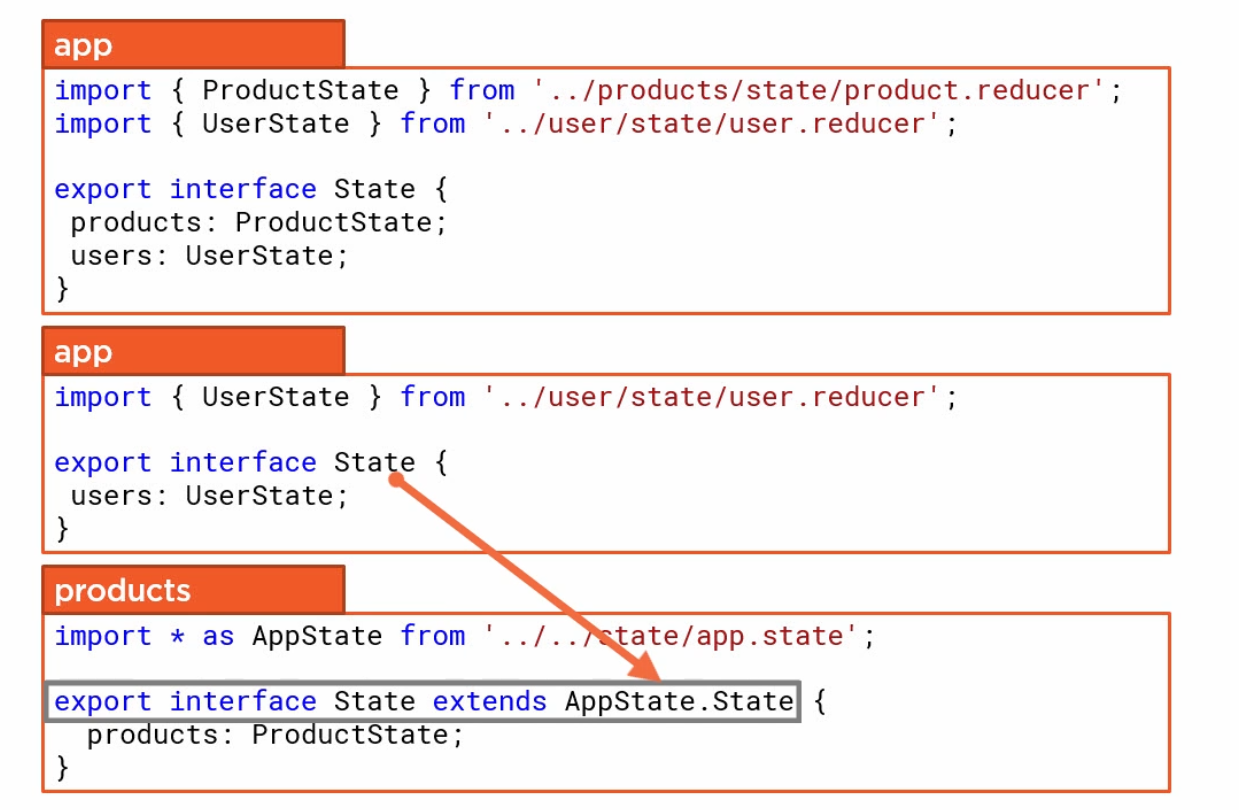


# Lazy loaded modules

If we define our state in that hierarchy it will break lazy loaded modules because we will reference them



We need to extend the state



If the module isn’t lazy loaded we can add it directly to State in App like UserState, but if it is lazy loaded we need to extend the state

# Selectors

A reusable query for the store. A level of abstraction between store and component

It is a function that returns a part of the state from the store

They are cached and only re-calculated if that part of state changes

If we change our Store structure we will only modify the selectors in one place instead of going to each component and modifying it

A selector is also a Pure Function

We can compose selectors and get a property that may not be defined in the state

## Benefits

* Provide strongly typed API
* Decouple the store from components
* Encapsulate complex data transformations
* reusable
* memoized

## Memoization

Every time selector is invoked with input, its result is cached so when its called a second time with same input, no need to re-execute and calculate the selector again. Instead, it returns the cached value

A selector's memoized value stays in memory indefinitely, until a new value replaces it

### Resetting a selector

If the memoized value is, for example, a large dataset that is no longer needed it's possible to reset the memoized value to null so that the large dataset can be removed from memory

selectTotal([state](https://ngrx.io/api/store-devtools/StoreDevtools" \l "state)); // returns the memoized [value](https://ngrx.io/api/store/testing/MockSelector" \l "value) of 8

selectTotal.release(); // memoized [value](https://ngrx.io/api/store/testing/MockSelector" \l "value) of selectTotal is now null

Releasing a selector also recursively releases any ancestor selectors.

[export](https://ngrx.io/api/store-devtools/DevToolsFeatureOptions" \l "export) const selectTotal = [createSelector](https://ngrx.io/api/store/createSelector)(   
selectSumEvenNums,  
selectSumOddNums,  
(evenSum, oddSum) => evenSum + oddSum);

Memoized values before release:   
selectSumEvenNums = 6  
selectSumOddNums = 4  
selectTotal = 10

After reset: all will be null

## Passing props to selectors

Definition

[export](https://ngrx.io/api/store-devtools/DevToolsFeatureOptions" \l "export) const getCount = [createSelector](https://ngrx.io/api/store/createSelector)(

getCounterValue,

(counter, [props](https://ngrx.io/api/store/props)) => counter \* props.multiply

);

Usage

this.store.select(getCount, { multiply: 2 })

using props with selectors disable the memoization property because every time the selector is invoked with new input, it is re-calculated

to overcome this we can add wrapper function around the selector as a factory method.

[export](https://ngrx.io/api/store-devtools/DevToolsFeatureOptions" \l "export) const getCount = () =>

[createSelector](https://ngrx.io/api/store/createSelector)(

([state](https://ngrx.io/api/store-devtools/StoreDevtools" \l "state), [props](https://ngrx.io/api/store/props)) => state.counter[props.id],

(counter, [props](https://ngrx.io/api/store/props)) => counter \* props.multiply

);

now the selector caller will always call this factory method to create different selector instances for each input combination. So when same selector is invoked with same input it will be memoized

## Using RxJs Pipes with selectors

We can use pipes with selectors for example to filter undefined values returned by state

store

.pipe(

map([state](https://ngrx.io/api/store-devtools/StoreDevtools" \l "state) => selectValues([state](https://ngrx.io/api/store-devtools/StoreDevtools" \l "state))),

[filter](https://ngrx.io/api/data/EntityCollection" \l "filter)(val => val !== undefined)

)

.subscribe(/\* .. \*/);

we can also use the built in select by NgRx

store

.pipe(

[select](https://ngrx.io/api/store/select)(selectValues),

[filter](https://ngrx.io/api/data/EntityCollection" \l "filter)(val => val !== undefined)

)

.subscribe(/\* .. \*/);

we can also define the pipe method as a selector and reuse it

[export](https://ngrx.io/api/store-devtools/DevToolsFeatureOptions" \l "export) const selectFilteredValues = pipe(

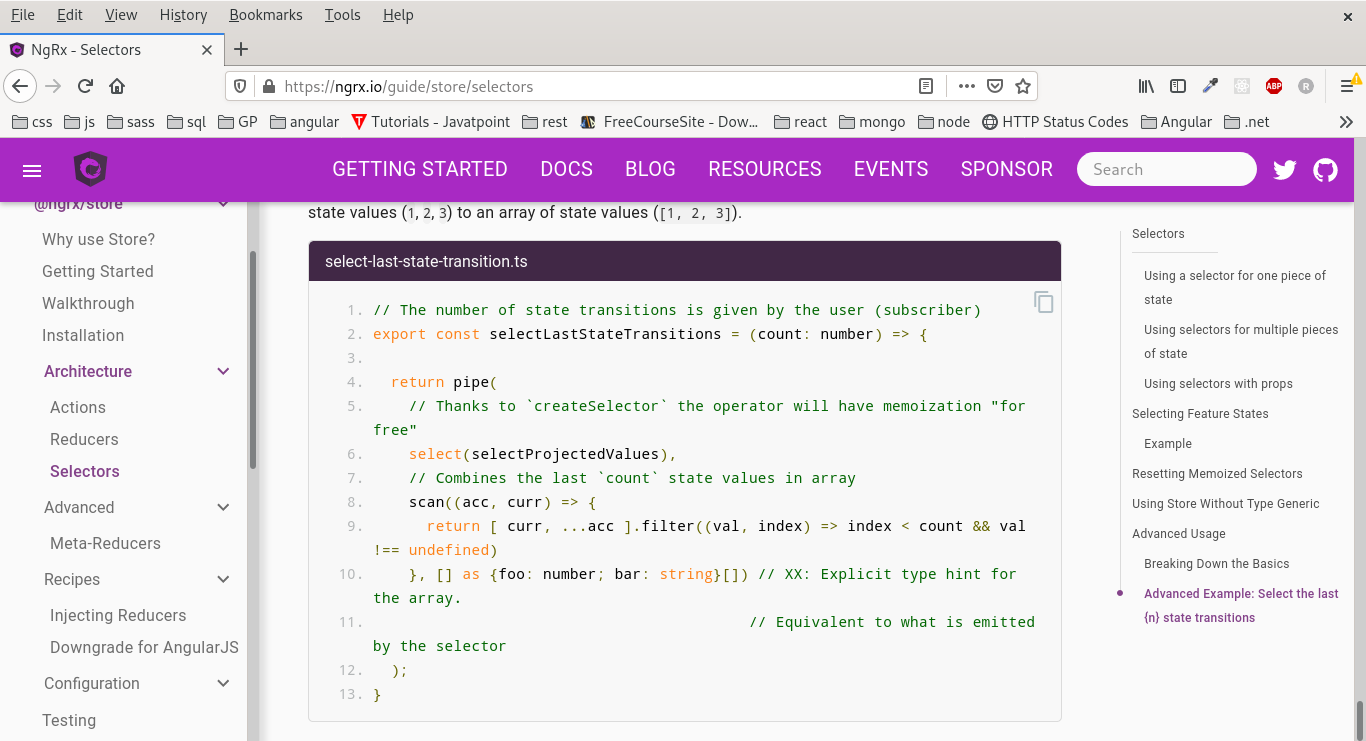
[select](https://ngrx.io/api/store/select)(selectValues),

[filter](https://ngrx.io/api/data/EntityCollection" \l "filter)(val => val !== undefined)

);

store.pipe(selectFilteredValues).subscribe(/\* .. \*/);

### Scan operator

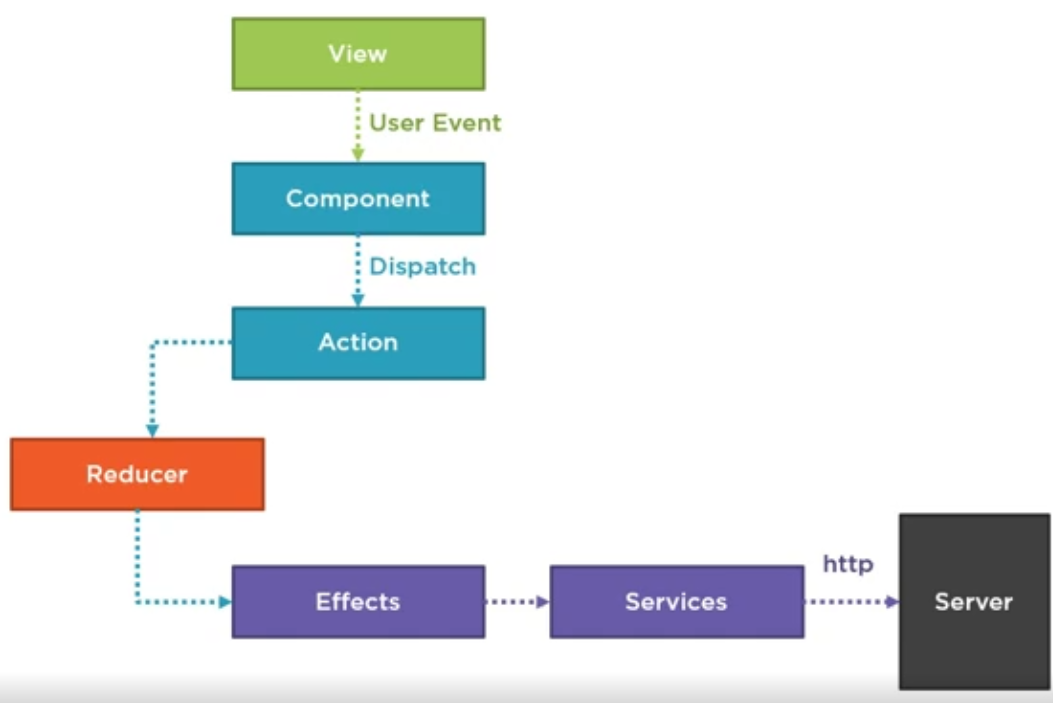
We can use scan operator to return history of state transitions. Like selecting n last state values

# Effects

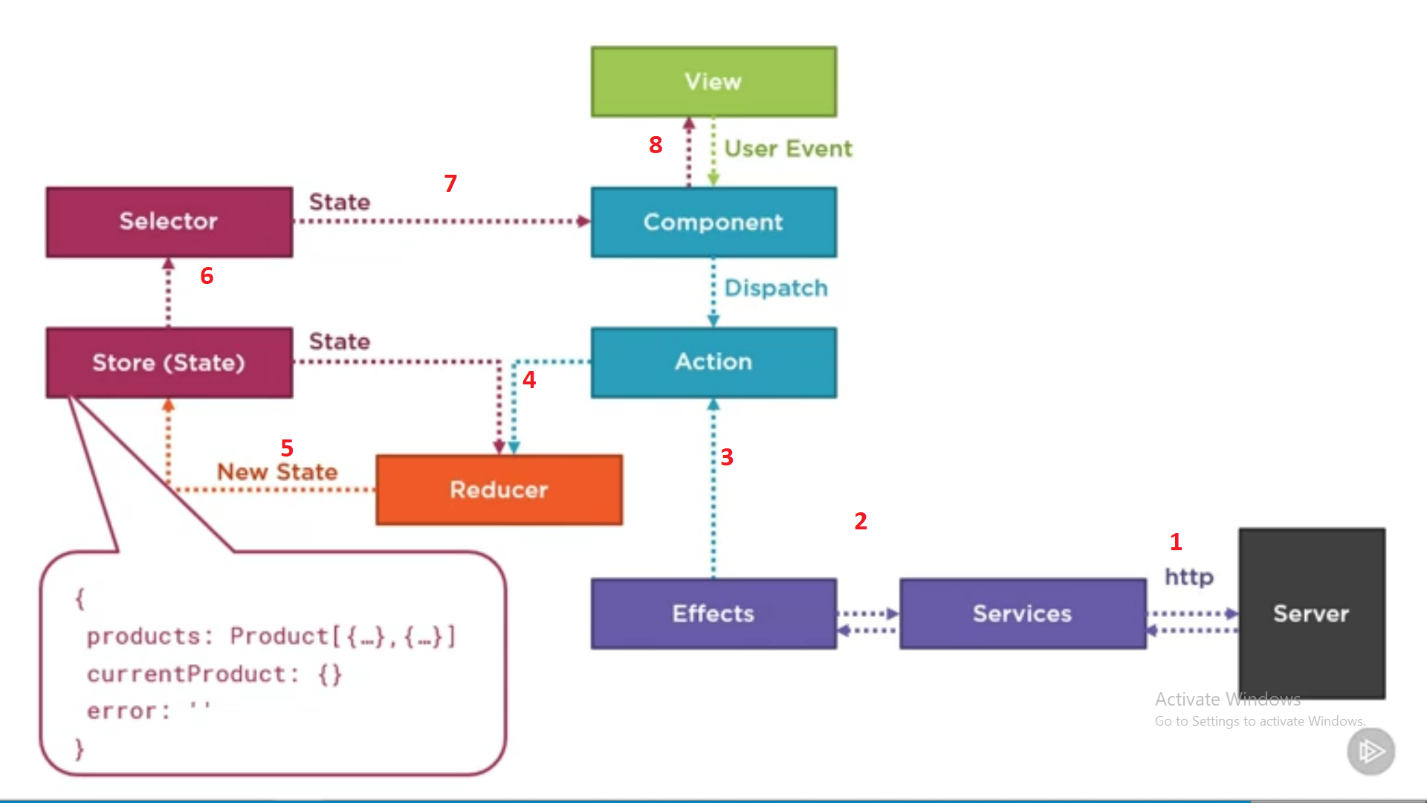
We cant make http requests inside our reducers because it is supposed to be a pure function without side effects. So instead of delegating this to the component, we can use Effects.

Effects take an action, do some work, dispatch a new action

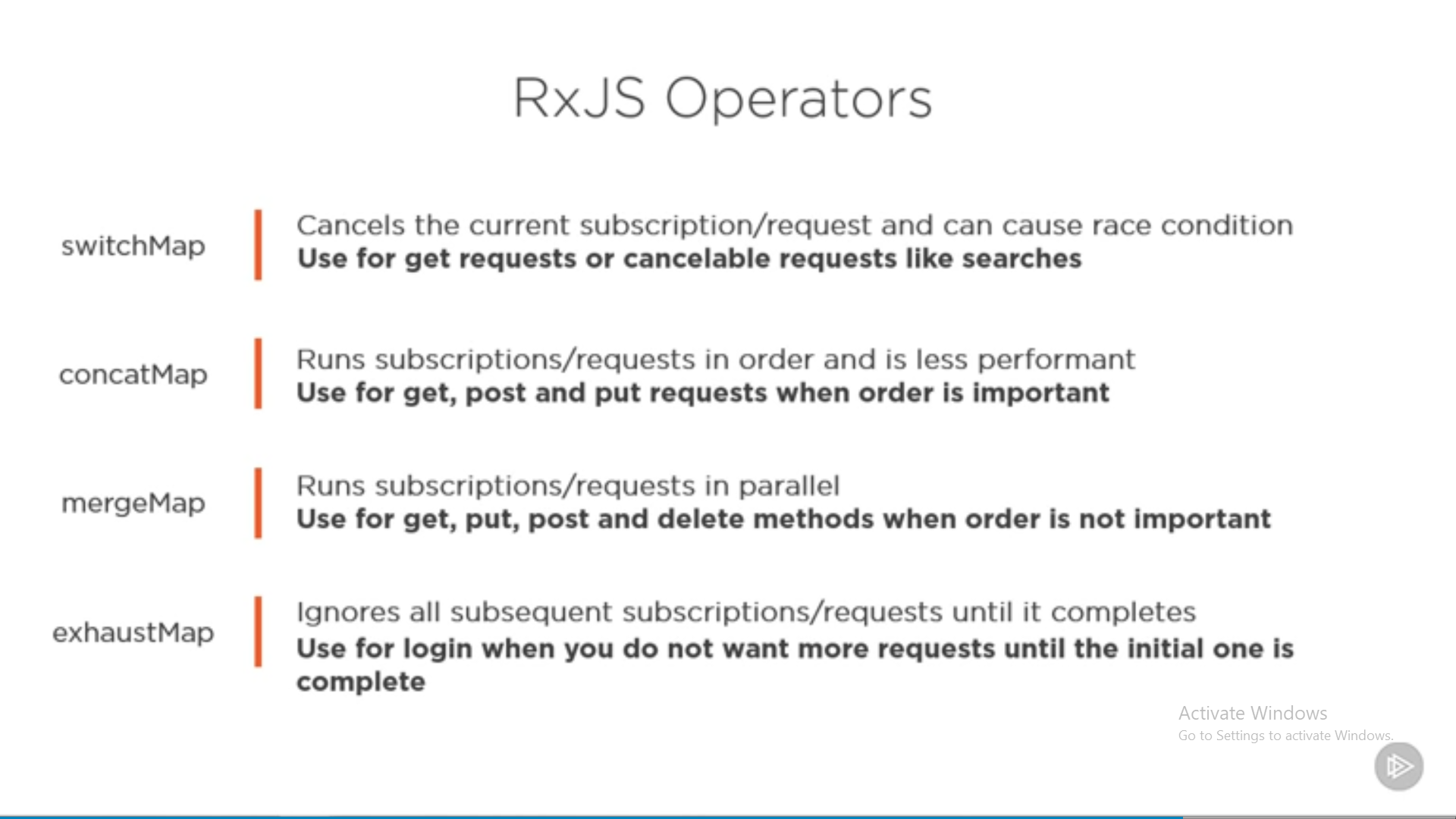
1. Component dispatches an action to the reducer (load products)
2. The reducer uses effects to get products
3. The effect calls the angular service which makes the http request



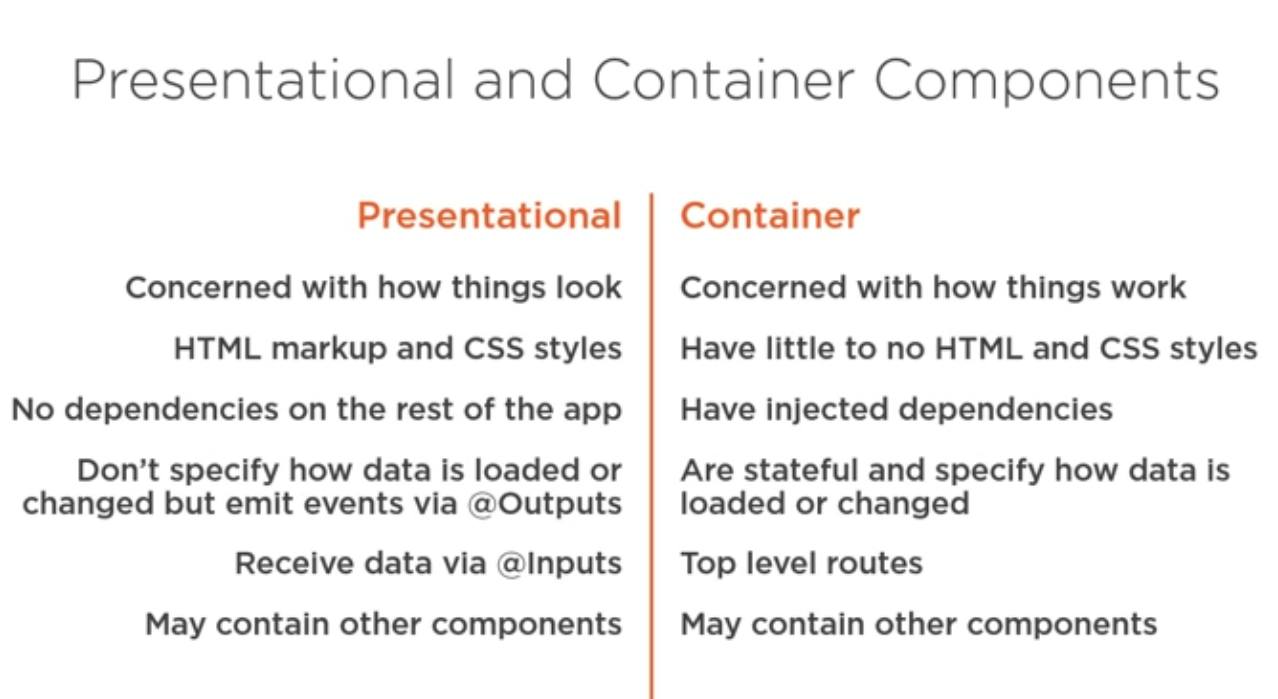
1. The response is passed from service to effect
2. The effect dispatches a new action (load products success) with the data retrieved from http request
3. The reducer add the results to the state
4. The component get notified when products list state changes



Which operators to use with effects and when



# Presentational vs Container components



## Presentational components pros

1. They have better performance because we can skip OnPush lifecycle method which allows to skip change detection on presentational components whose inputs haven’t changed
2. Easier to compose into smaller components and reuse them
3. Easier to test

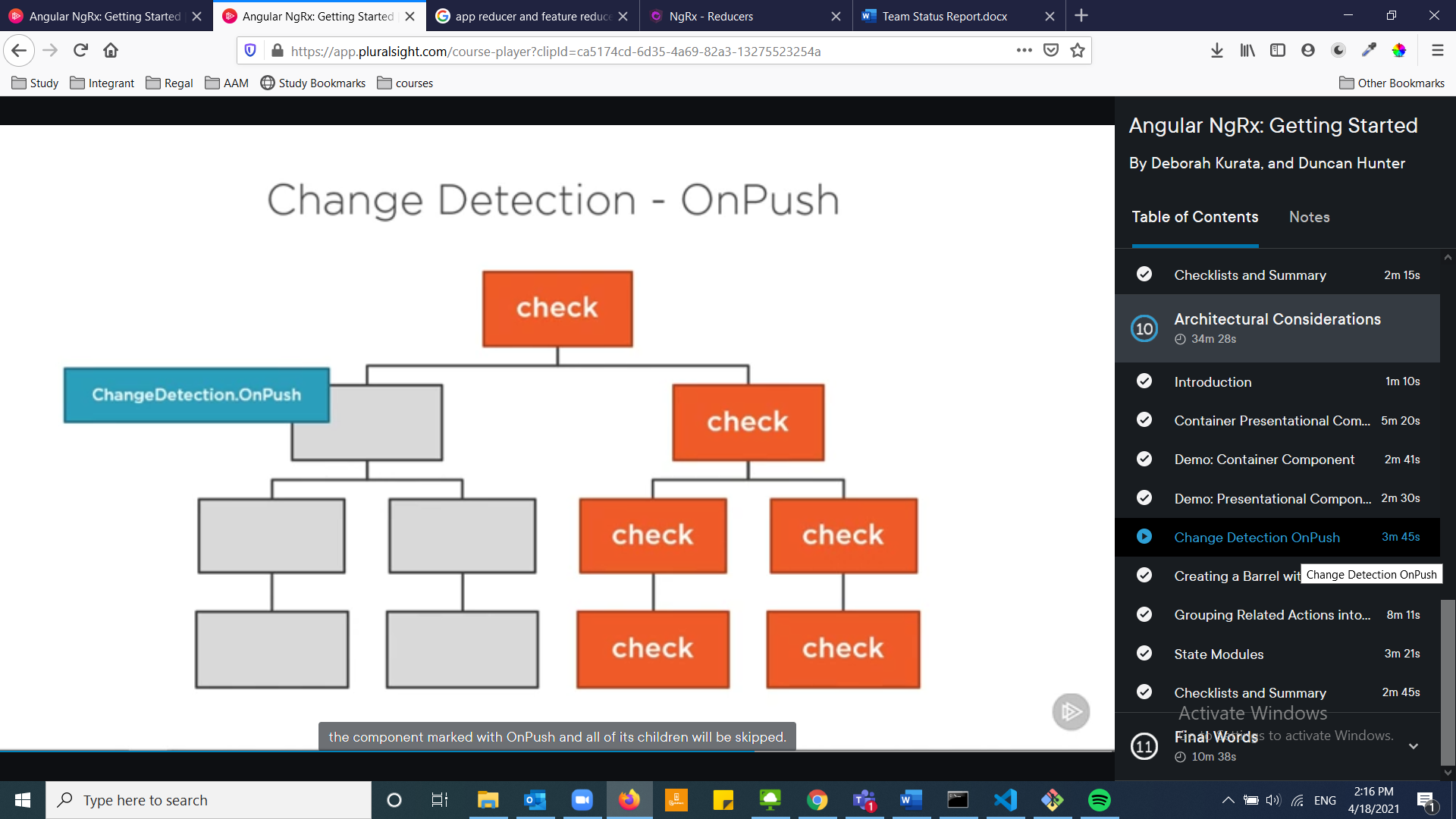
## Change detection Strategy

### Default

Everytime something changes in app as a result of timer or http request or promise, change detection will run on all components

If an event fired in a child component, angular will start checking for change detection from root component down to every child component.

### OnPush



In case of OnPush, the component marked with OnPush and all its children will be skipped. Unless it receives a new input value or object reference changed. But it doesn’t detect if a property in object has changed

# Notes:

* Its better to keep components presentational if possible
* Using OnPush with presentational components
* Create index.ts and export everything needed (Actions, Selectors, State)
* Actions should capture events not commands
  + Split actions names by page then by api type
    - [Product Page] Load Products: fired from user event
    - [Product API] Load Success: interact with api